A parallel ILP algorithm that incorporates incremental batch learning

Nuno Fonseca[1], Rui Camacho[2], and Fernando Silva[1]

[1] DCC-FC & LIACC, Universidade do Porto
R. do Campo Alegre 823, 4150-180 Porto, Portugal
{nf, fds}@ncc.up.pt
[2] Faculdade de Engenharia & LIACC, Universidade do Porto
Rua Dr. Roberto Frias, s/n 4200-465 Porto, Portugal
rcamacho@fe.up.pt

Abstract. In this paper we tackle the problems of efficiency and scalability faced by Inductive Logic Programming (ILP) systems. We propose the use of parallelism to improve efficiency and the use of an incremental batch learning to address the scalability problem. We describe a novel parallel algorithm that incorporates into ILP the method of incremental batch learning. The theoretical complexity of the algorithm indicates that a linear speedup can be achieved.

Key words: ILP, Parallelism, Incremental batch learning, Scaling-up

1 Introduction

Inductive Logic Programming (ILP) [1] is an established subfield of Machine Learning (ML). The objective of ILP is the induction of first-order clausal theories from a set of examples and prior knowledge. There are two major motivations for using ILP. First, ILP provides an excellent framework for learning in multi-relational domains. Second, the theories learned by general purpose ILP systems are in a high-level formalism, which is often understandable and meaningful for the domain experts. The advantages of ILP have been demonstrated through successful applications in difficult, industrially and scientifically relevant problems. An up-to-date list of applications of ILP systems to real world problems can be found in [2].

One major criticism of ILP systems is that they often have long running times. Research in improving the efficiency of ILP systems has been focused in reducing their sequential execution time, either by reducing the number of hypotheses generated (see, e.g., [3,4]), or by efficiently testing candidate hypotheses (see, e.g., [5,6,7]). Another way of improving the response time of ILP systems, besides improving their sequential efficiency, is through parallelization. As pointed out by Page [8], and confirmed by research results [9,10,11,12], parallelization of ILP systems is a research direction that should be pursued further.

This paper addresses the scalability and efficiency problems of ILP systems. We propose a parallel procedure that tackles both problems simultaneously. The procedure explores the idea of synthesizing models from subsets of the training data in parallel. The construction of the models is incremental. At each step, one of the subsets is used to generate a model that will be used in the next step. Such scheme can be seen as a sort of incremental batch learning [13,14] for ILP. An algorithm that incorporates both incremental batch learning and parallelism is presented and discussed.

The remainder of this paper is organized as follows. Section 2 provides some ILP background, including a description of an ILP procedure. Section 3 presents and discusses a parallel scheme that explores the idea of synthesizing models from subsets of data in parallel. Next, in Section 4 we describe related work. We draw some conclusions in Section 5 and point out future work.

2 ILP

This section briefly presents some concepts and terminology of Inductive Logic Programming but is not intended as an introduction to the field of ILP. For such introduction we refer to [15,16].

2.1 Problem

The objective of an ILP system is the induction of logic programs. As input an ILP system receives a set of examples (divided in positive and negative examples) of the concept to learn, and sometimes some prior knowledge (or *background knowledge*). Both examples and background knowledge are usually represented as logic programs. An ILP system tries to produce a logic program where positive examples succeed and the negative examples fail.

From a logic perspective, the ILP problem can be defined as follows. Let E^+ be the set of positive examples, E^- the set of negative examples, $E = E^+ \cup E^-$, and B the background knowledge. In general, B and E can be arbitrary logic programs. The aim of an ILP system is to find a set of hypotheses (also referred to as a theory) H, in the form of a logic program, such that the following conditions hold:

- **Prior Satisfiability**: $B \wedge E^- \nvDash \square$
- **Prior Necessity**: $B \nvDash E^+$
- **Posterior Satisfiability**: $B \wedge E^- \wedge H \nvDash \square$ (Consistency)
- **Posterior Sufficiency**: $B \wedge H \vDash E^+$ (Completeness)

The sufficiency condition is sometimes named *completeness* with regard to positive evidence, and the posterior satisfiability is also known as *consistency* with the negative evidence.

In short, the problem of ILP is to find a consistent and complete theory, i.e., a set of clauses that "explain" all given positive examples and is consistent with

the given negative examples. Since it is not immediately obvious which set of clauses should be picked as the theory, an ILP system performs a search through the permitted clauses to find a set with the desired properties.

To find a satisfactory theory, an ILP system searches through a search space of the permitted clauses. The states in the search space (designated as *hypothesis space*) are concept descriptions (hypothesis) and the goal is to find one or more states satisfying some quality criterion. The hypotheses generated during the search are evaluated to determine their quality. A widely used approach to score an hypothesis is by computing its accuracy and coverage. The *accuracy* is the percentage of examples correctly classified by an hypothesis. The *coverage* of an hypothesis h is the number of positive (*positive cover*) and negative examples (*negative cover*) derivable from $B \wedge h$. The time needed to compute the coverage of an hypothesis depends primarily on the cardinality of E^+ and E^-.

2.2 Mode-Directed Inverse Entailment

Mode-Directed Inverse Entailment [17] (MDIE) is a technique widely implemented in ILP systems that uses inverse entailment together with mode restrictions to find a hypotheses set H.

---

induce(B,C,E)
Input: Background knowledge B, hypothesis constraints C, and a finite training set $E = E^+ \cup E^-$.
Output:A set of hypotheses complete and consistent.

1. i=0
2. H_i=$\varnothing$
3. if E^+=$\varnothing$ return H_i
4. increment i
5. $Train$=$E^+ \cup E^-$
6. e_i^+=select an example from E^+
7. $\perp_i$=$saturate(e_i, B, H_{i-1}, Train, C)$
8. h_i=$search(B, H_{i-1}, C, Train, e_i, \perp_i)$
9. H_i=$H_{i-1} \cup h_i$
10. $E_{covered}$=$\{e \mid e \in E^+ \wedge B \cup H_i \vDash e\}$
11. E=$E^+ \setminus E_{covered}$
12. Go to Step 3

Fig. 1. An induction procedure based on MDIE. The saturation procedure constructs the most-specific-clause $\perp_{(e_i,B,H_{i-1},E,C)}$ that entails the example selected, and is within language restrictions provided (C). The most-specific-clause, also called "bottom clause" [17], is usually a definite clause with many literals. The search procedure invoked in step 8 finds the best consistent clause more general than example e_i.

The procedure presented in Fig. 1 induces a theory H from a set of examples E, background knowledge B, and some constraints C, by following a greedy cover set approach. Variants of this procedure are implemented by a number of ILP systems (such as [18,19,7]). The `search` procedure invoked in step 8 needs to be described. The search is done by performing a general-to-specific search in the subsumption lattice bounded bellow by $\perp_{(e_i,B,H,E,C)}$. The clauses' bodies generated during the search are subsets of the literals from the bottom clause. One of the constraints in C imposes a limit on the number of hypotheses generated, thus ensuring that the search terminates. If no hypothesis is found by the `search` procedure then the example e_i is returned, and it is said that e_i is incompressible with the input provided to `search`.

The step 7 is designated as saturation step, step 8 is sometimes designated as reduction step, and steps 10 and 11 are the cover removal step. Thus the procedure can be simplified as: 1) select example; 2) saturate; 3) reduce; and 4) remove covered examples.

2.3 Time complexity

In this section we are concerned with time complexity of the procedure described above regarding the number of steps executed. Before proceeding we need to introduce the following constants. Let `I`[3] be the number of iterations of the loop in the procedure, $n_\oplus$ and $n_\ominus$ the number of positive and negative examples, `n` the total number of examples (i.e., $n_\oplus + n_\ominus$), and `S` be the limit on the number of hypothesis generated during the search. The value of S is usually bounded by the user (when unbound it would be proportional to the cardinality of $|\perp|$).

The cost of step 6 is naturally dependent on the implementation and type of example's selection (e.g., random or sequential). We will consider that it can be done by accessing the set of examples in a constant time ($C1$). The cost of the `saturate` procedure is directly proportional to the cardinality of $\perp$ [17]. We will consider that saturation can be done in Sat steps, for some value Sat proportional to the maximum cardinality of all $\perp_i$. The `search` procedure mainly generates and tests hypotheses, thus the cost associated should be $S(Gen+Eval*n)$ where Gen and $Eval$ are, respectively, the cost of generating a hypothesis and testing it against an example. Note that the cost of evaluating an hypothesis against an example can be extraordinary high specially if the background knowledge contains highly non-deterministic predicates. The removal of the examples covered (steps 10 and 11) can be done, in the worst case, in $n_\oplus$ steps. The costs of the other steps are negligible, and thus not considered.

The cost required to execute the `induce` procedure can be described as

$$T_{induce} = I(C1 + Sat + S * (Gen + n * Eval) + n_\oplus))$$

Taking into account that $I \leq n$ and Sat is smaller than $S * (Gen + n * Eval)$ then T_{induce} is $O(S * n^2)$. Note that we left S in because S can be much larger than n.

[3] I may take values ranging from 1 to $\mid E^+ \mid$. The greatest value occurs when the search procedure is unable to "compress" all examples $e \in E^+$.

3 Parallel model synthesis from subsets of the training data

To explore the idea of synthesizing models from subsets of training data we propose to incorporate Incremental Batch Learning [13,14] into ILP. In Incremental Batch Learning (see Fig. 2) the set of examples E is randomly divided into a partition of p subsets of approximately equal size (maintaining the proportion of positive and negative examples). A learning procedure is applied to a subset of examples E_i to build a C_i using C_{i-1}. By breaking up the set of examples into p subsets, the process of learning an hypothesis is broken into a sequence of p processes ("pipeline") such that once the first process is completed the output is passed as input to the second, and so on, until all p processes are executed.

Fig. 2. Incremental Batch Learning

Our proposal to incorporate Incremental Batch Learning into ILP consists in pipelining the reduction step. The idea is to use a modified version of the search procedure (described previously) as the "learning procedure". The modified search procedure (`search'`) should accept a set of candidate hypotheses (C_{i-1}), as an extra input argument, that determine the points in the generalization lattice from where the search will start. The output of the `search'` procedure is a set of candidate hypotheses (C_i) consistent with (E_i). The initial set of hypotheses (C_0) is the empty set. The best hypothesis in C_p should correspond to the hypothesis h_i in Fig. 1.

The Fig. 3 illustrates how Incremental Batch Learning works to find a range definition in a numerical domain. An example is selected, saturated, and then a search for an hypothesis is performed. A set of candidate hypotheses C_1 is generated using E_1 as the set of examples. These candidate hypotheses are further revised using E_2. The best hypothesis in C_2 (`r(X):-X>=1,X=<4`) is selected as h_i at the end of the "pipeline".

In the remaining of the section we present and discuss a parallel procedure that incorporates Incremental Batch Learning into ILP. A cost analysis of the procedure is also provided.

Fig. 3. Sequential incremental batch learning in ILP

3.1 Logical setting

Based on the ILP setting presented in Section 2.1 we now formalize the logical setting of learning from subsets of the training data.

Let $E_1, \ldots, E_p$ be a partition of the examples into p subsets such that $(\cup_{t=1}^{p} E_t = E)$ and $(\cap_{t=1}^{p} E_t = \varnothing)$. The task of learning an hypothesis is thus divided into p sub-tasks, in which each subtask t induces a set of hypotheses C_t from E_t. Each hypothesis h in C_t is valid in E_t, but may not be consistent in E. The hypotheses in C_t are incrementally built from C_{t-1}. The incremental learning process should ensure:

- Incremental completeness: $B \cup h \models \bigcup_{i=1}^{t} E_i^+$
- Incremental consistency: $B \cup h \not\models \bigcup_{i=1}^{t} E_i^-$

From incremental consistency and completeness conditions we know that each hypothesis in C_t is complete and consistent with E.

3.2 Parallel algorithm

We now describe the parallel procedure illustrated in Fig. 4. The underlined lines correspond to subtasks that are executed in parallel. This procedure extends the sequential procedure presented in Fig. 1 by incorporating incremental batch learning and parallelism.

The procedure starts by randomly creating p mutually exclusive subsets $E_1, \ldots, E_p$ of E of approximately equal size, where p is also the number of processors available. Each subset E_k is attributed to one processor together with other data (such as the background knowledge). The processor that partitions the data is designated as the master. The other processors (slaves) execute subtasks delegated by the master or by other processors.

In steps 10 and 11 the master selects one example e_k from E_k and generates $\bot_k$ by saturating the example. In step 12 the master delegates on processor k the task of searching for a set of candidate hypotheses C^k using $\bot_k$. The processor k then starts the process of incremental batch learning by invoking the `search'` procedure in all p processors (starting in processor k). At each stage, a processor j $(1 \leq j \leq p)$ executes the procedure `search'` to build C_j^k using C_{j-1}^k. The last

p-ibl-induce(B,C,E,NCS,p)
Input: Background knowledge B, hypothesis constraints C, a finite training set $E = E^+ \cup E^-$, the number of processors p available, and the maximum number of consistent hypotheses NCS returned by search.
Output:A set of complete and consistent hypotheses.

1. $i = 0$, $EP =\mid E^+ \mid$, $H_i = \varnothing$
2. $< E_1, \ldots, E_p >$=partition of E in p subsets
3. send E_k, B, and C to each processor k
4. if $EP = 0$ return H_i
5. increment i, H_i=H_{i-1}
6. $Train_{ij} = E_j^+ \cup E_j^-$ on each processor j ($j = 1, \ldots, p$)
7. for proc=1 until p
8. k=(proc+1)%p
9. $C_{k-1}^k = \varnothing$
10. e_k^+=select an example from E_k^+
11. $\perp_k$= $saturate(e_k, B, H_i, E, C)$
12. $C^k = search'(B, H_i, C, Train_{ij}, e_k^+, \perp_k, C_{j-1}^k)$ on each processor j ($j = 1, \ldots, p$) starting in k
13. $compute\_coverage(C^k, E_j)$ on each processor j ($j = 1, \ldots, p$)
14. end for
15. (synchronize)
16. $Candidates$=$\bigcup_{j=1}^{p} clean(C^j)$
17. repeat
18. h_i =select best from $Candidates$
19. for j=1 until p run in parallel on processor j
20. $H_i = H_i \cup h_i$
21. $E_{covered}^j = \{e \mid e \in E_j^+ \wedge B \cup H_i \models e\}$
22. $E_j = E_j^+ \setminus E_{covered}^j$
23. $compute\_coverage(Candidates, E_j)$
24. end for
25. (synchronize)
26. remove hypotheses from $Candidates$ that are not complete
27. $EP = EP - \sum_{j=1}^{p} \mid E_{covered}^j \mid$
28. if $Candidates = \varnothing$ break
29. end repeat
30. Go to Step 4

Fig. 4. A parallel incremental batch learning procedure. The subtasks underlined can be executed in parallel. The `search'` procedure invoked in step 12 is a modified version of the procedure described in Section 2.2. The procedure `compute_coverage(`S`,`E_k`)` computes the coverage of a set of hypothesis S on a processor k using the subset E_k and "places" the results in the master. The `clean` procedure in step 16 removes the hypotheses that explain only one example but does not remove hypotheses that are examples.

processor in the pipeline sends the set of candidate hypotheses found to the master. Fig. 5 shows how the search would be performed using two processors for the example given in the beginning of this section. The master (processor 1) selects an example, saturates it, and then initiates the search process in the processor 2 (using the subset E_2) to build C_1^2, that is then passed to the next processor in the pipeline (processor 1) to build C_2^2. Meanwhile, the master selected and saturated another example and started the search process locally using the subset E_1 to build C_1^1. Once built, C_1^1 is passed to processor 2 to build C_2^1. The values of C^1 and C^2 correspond respectively to `r(X):-X=<4,X>=1` and `r(X):-X>=1,X=<4`. In step 13 the positive coverage of the candidate hypotheses in C^k is reevaluated on all processors to obtain global accuracies.

Fig. 5. Parallel Incremental Batch Learning in ILP

A synchronization point occurs at step 15. The master only proceeds to step 16 when all set of candidate hypotheses C^j $(1 \leq j \leq p)$ have been received. In steps 17 to 29 the best hypotheses found are added to the theory (H_i). A criteria to select the hypotheses from *Candidates* is described in the next subsection. The best hypothesis is selected, added to the theory, and the examples explained by it removed from each E_j. The positive coverage of the remaining candidate hypotheses is reevaluated against each E_j.

The described procedure is not complete with relation to the procedure presented in Fig. 1 because the best hypothesis found using a subset of examples may not be the best hypothesis in the search space if all examples available were considered. To improve the procedure completeness we need to make a change to the `search` procedure. It should return all consistent hypotheses in the search space, and not only the best one. The *NCS* parameter can be used to bound the number of consistent hypotheses that each C_t may contain. If this value is unbound, then all consistent hypotheses are returned.

Candidate hypotheses selection An obvious option to order the candidate hypotheses would be the use of their global accuracy. However, we also have the possibility of obtaining partial accuracies, one for each subset of examples E_i. An hypothesis h has an accuracy on partition E_k of $(acc(h,k) = \frac{c}{\mid E_k \mid})$ where c is the number of examples in E_k that are correctly classified by h. Note that the global and partial accuracies are an estimate of the real accuracy of h that is usually unknown.

One advantage of the presented procedure is that it may provide better accuracy estimators. By generating an hypothesis h in a subset i and later evaluating it in other subsets we obtain partial accuracies $acc(h,1), \ldots, acc(h,p)$. An accuracy estimator of h to be unbiased must be evaluated in data selected independently from h [20]. The subsets E_k (where $k \in \{k \mid 1 \leq k \leq p \wedge k \neq i\}$) are relatively independent from h and provide a relative unbiased accuracy estimator. From an unbiased estimator we obtain a better approximation of the real accuracy of h by calculating the mean accuracy as follows.

$$\widehat{\mu} = \frac{1}{p-1} \sum_{j=1, j \neq i}^{p} acc(h,j) \tag{1}$$

3.3 Time complexity

In this section we provide a preliminary analysis of the time complexity of the proposed procedure. The variables introduced in Section 2.3 will be reused next. The cost of some steps are negligible and for that reason are not considered.

Parallel algorithms contain some sources of overhead such as communication time for sending messages, extra computations required for the parallel code, and periods when the processors are idle. Since most of these costs are implementation dependent we do not take them into account in our analysis. Nevertheless, we believe that this simplification is compensated by an overestimation of the computation effort. For instance, we did not take into account the decrease in the number of positive examples verified during each iteration of i, when in fact it should decrease at least p examples in each iteration.

Step 3 creates p subsets of E. Such operation would require to access n examples. Sending the background knowledge, a partition of examples and the constraints should require a $C2$ amount of time.

Steps 7 to 14 contains a loop that performs p times: select an example, saturate example, execute reduction in parallel, and reevaluates all C^k found in each processor in parallel. The cost of selection and saturation was defined previously as $C1$ and Sat respectively. The cost of the reduction step is more complicated to estimate, nevertheless we present a pessimistic estimate of $p * S * (Gen + \frac{n * Eval}{p})$. The cost of reevaluating all candidate hypotheses is estimated in $\frac{NCS * n_{\oplus} * Eval}{p}$. Hence, the total cost of the loop is $p * (C1 + Sat) + p * S * (Gen + \frac{n * Eval}{p}) + p * \frac{NCS * n_{\oplus} * Eval}{p}$. The cost of the step 16 is $NCS * p$.

In steps 17 to 29 a subset of the $Candidate$ hypotheses is added to the theory, one hypothesis at a time. First, the best hypothesis is selected from $Candidates$

whose cardinality is, in the worst case, $NCS*p$. Next, cover removal is performed in parallel on each subset together with a new reevaluation of the remaining candidate hypotheses. The cost of these steps is $p*(NCS*p+\frac{n_{\oplus}}{p}+\frac{Eval*n_{\oplus}}{p})$, assuming that the body of the loop is executed p times.

Taking into account that p processors are being used, the number of iterations of the outer loop is $\frac{I}{p}$. The number of steps estimated to execute the `p-ibl-induce` procedure is

$$n+C2+I(C1+Sat)+I*S*Gen+\frac{I*S*n*Eval}{p})+\frac{I*NCS*n_{\oplus}*Eval}{p}+I*NCS(1+p)+I\frac{n_{\oplus}}{p}(1+Eval)$$

Following the explanations given in Section 2.3 together with $T_{p-ibl-induce}$ we deduce that the complexity of the procedure is $O(\frac{S*n^2}{p}))$.

The speedup is computed as the ratio of T_{induce} by $T_{p-ibl-induce}$ and is $O(p)$. These values are only an estimate and the final value will depend on the implementation and on the dataset used. However, it indicates that a linear speedup could be achieved.

4 Related Work

Several approaches have already been implemented to parallelize ILP systems. We describe in this section some of those approaches and relate them with our proposal. We also relate our work to some approaches studied in ILP that learn from a partition of the data.

The strategies used so far to parallelize ILP systems can be categorized as: parallel exploration of independent hypotheses [10]; parallel exploration of the search space [10,11]; parallel coverage test [10,12]; and parallel execution of an ILP system over a partition of the data [9,12]. Our proposal incorporates three of the four strategies used so far to parallelize ILP systems. Data partitioning is performed by distributing the set of examples among all processors where a modified version of the reduction step is executed. It also performs parallel coverage tests of the hypotheses found to be globally consistent. Finally, it explores the search space in parallel as each processor starts by exploring a search space bounded bellow by a bottom clause generated from an example of its subset.

Learning from subsets of data has also deserved some attention from the ILP community. A procedure called layered learning that constructs, in stages, increasingly correct theories is described in [21]. The procedure starts by using a small sample of the data to construct an approximately correct theory, that is improved in the next stages. The sample at each stage is a superset of the sample of the previous stage. This is a major difference in relation to our approach. Subsampling and logical windowing was studied in the ILP context by Srinivasan [22]. A main difference from logical windowing to our proposal is that in windowing the sample is a superset of the sample of the previous stage, while our approach partitions the sample into independent subsets and learning is performed at each stage using a single subset. Subsampling consists in repeating

the holdout method k times, and the estimated accuracy is derived by averaging the runs. Our approach is orthogonal to subsampling, since it can be applied to the training set generated by the subsampling procedure.

5 Conclusions and future work

This work proposed a parallel procedure for improving the efficiency and scalability of ILP systems. The proposed scheme explores the idea of synthesizing models from subsets of the training data in parallel. The construction of the models is made using a pipeline strategy. At each stage, one of the subsets is used to generate a model that will be refined in the subsequent stages. The scheme can be seen as a kind of incremental batch learning. A procedure that exploits this scheme using a multi-processor environment was presented and analyzed.

This paper has two major contributions. First, we incorporate incremental batch learning into ILP. Secondly, we describe a novel ILP parallel procedure (based on incremental batch learning) that may scale up and and improve ILP systems efficiency. The parallel procedure has an expected linear speedup. If an implementation confirms this then the natural consequence is that ILP systems may start to address problems that otherwise would be inaccessible. Our work has some shortcomings. First, the proposed parallel procedure is specific to MDIE based ILP systems. Secondly, the syntheses of models from subsets as proposed is not complete regarding a conventional MDIE based procedure.

In the immediate future we plan to implement and evaluate our proposal.

Acknowledgments

The work reported has been partially supported by the project APRIL (Project POSI/SRI/40749/2001) and funds granted to *LIACC* through the *Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia* and *Programa POSI*. Nuno Fonseca is funded by FCT grant SFRH/BD/7045/2001.

References

1. S. Muggleton. Inductive logic programming. In *Proceedings of the 1st Conference on Algorithmic Learning Theory*, pages 43–62. Tokyo, Japan, 1990.
2. Ilp applications. http://www.cs.bris.ac.uk/ ILPnet2/Applications/.
3. C. Nédellec, C. Rouveirol, H. Adé, F. Bergadano, and B. Tausend. Declarative bias in ILP. In L. De Raedt, ed., *Advances in Inductive Logic Programming*, pages 82–103. IOS Press, 1996.
4. Rui Camacho. Improving the efficiency of ILP systems using an incremental language level search. In *Annual Machine Learning Conference of Belgium and the Netherlands*, 2002.
5. Hendrik Blockeel, Luc Dehaspe, Bart Demoen, Gerda Janssens, Jan Ramon, and Henk Vandecasteele. Improving the efficiency of Inductive Logic Programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166, 2002.

6. Vítor Santos Costa, Ashwin Srinivasan, Rui Camacho, Hendrik Blockeel, and Wim Van Laer. Query transformations for improving the efficiency of ilp systems. *Journal of Machine Learning Research*, 2002.
7. Rui Camacho. *Inducting models of human control skills unsing ML algorithms*. PhD thesis, Univerity of Porto, 2000.
8. David Page. ILP: Just do it. In J. Cussens and A. Frisch, eds., *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 3–18. Springer-Verlag, 2000.
9. L. Dehaspe and L. De Raedt. Parallel inductive logic programming. In *Proceedings of the MLnet Familiarization Workshop on Statistics, Machine Learning and Knowledge Discovery in Databases*, 1995.
10. Hayato Ohwada and Fumio Mizoguchi. Parallel execution for speeding up inductive logic programming systems. In *Lecture Notes in Artificial Intelligence*, number 1721, pages 277–286. Springer-Verlag, 1999.
11. Hayato Ohwada, Hiroyuki Nishiyama, and Fumio Mizoguchi. Concurrent execution of optimal hypothesis search for inverse entailment. In J. Cussens and A. Frisch, eds., *Proceedings of the 10th International Conference on Inductive Logic Programming*, volume 1866 of *Lecture Notes in Artificial Intelligence*, pages 165–173. Springer-Verlag, 2000.
12. Y. Wang and D. Skillicorn. Parallel inductive logic for data mining. In *Workshop on Distributed and Parallel Knowledge Discovery, KDD2000*, Boston, 2000. ACM Press.
13. Clearwater S. H., Cheng T. P., Hirsh H., and Buchanan B. G. Incremental batch learning. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 366–370, San Mateo, CA, 1989. Morgan Kaufmann.
14. Foster J. Provost and Venkateswarlu Kolluri. A survey of methods for scaling up inductive algorithms. *Data Mining and Knowledge Discovery*, 3(2):131–169, 1999.
15. S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20:629–679, 1994.
16. S.-H. Nienhuys-Cheng and R. de Wolf. *Foundations of Inductive Logic Programming*, volume 1228 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, February 1997.
17. S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.
18. Stephen Muggleton and John Firth. Relational rule induction with cprogol4.4: A tutorial introduction. In Saso Dzeroski and Nada Lavrac, eds. , *Relational Data Mining*, pages 160–188. Springer-Verlag, September 2001.
19. Aleph. http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/.
20. T. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.
21. S. Muggleton. Optimal layered learning: A PAC approach to incremental sampling. In K. Jantke, S. Kobayashi, E. Tomita, and T. Yokomori, eds, *Proceedings of the 4th Conference on Algorithmic Learning Theory*, pages 37–44. Springer-Verlag, 1993.
22. A. Srinivasan. A study of two sampling methods for analysing large datasets with ILP. *Data Mining and Knowledge Discovery*, 3(1):95–123, 1999.